UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Automatización mediante VUNIT de la verificación de microprocesadores RISC-V descritos en HDL.

Trabajo Fin de Grado

Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación

AUTOR/A: Sánchez Alfaro, Marc

Tutor/a: Gadea Gironés, Rafael

Cotutor/a: Monzó Ferrer, José María

CURSO ACADÉMICO: 2021/2022

Resumen

El presente trabajo muestra como verificar un core IP de procesador RISC-V en VUnit y que soporte de forma parcial las instrucciones RV32I en su versión Single-Cycle. Además de mostrar los posibles errores que puedan ocurrir en la versión Multi-Cycle si se realiza una mala segmentación o se tratan los errores de manera incorrecta.

Se va a detallar el funcionamiento del core IP como caja negra, así como el proceso de instalación de VUnit, su funcionamiento y sus ventajas frente a otros tipos de verificaciones. También se van a mostrar las etapas de verificación que se han ido siguiendo hasta conseguir verificación completa, desde la más sencilla hasta la más compleja.

También se va a detallar la realización de una interfaz de usuario para poder ejecutar la propia verificación de una manera más intuitiva y práctica. Además de que archivos son los ejecutables de la misma y su uso.

Resum

El present treball mostra com verificar un core IP de processador RISC-V en VUnit i que suport de manera parcial les instruccions RV32I en la seua versió Single-Cycle. A més de mostrar els possibles errors que puguen ocórrer en la versió Multi-Cycle si es realitza una mala segmentació o es tracten els errors de manera incorrecta.

Es detallarà el funcionament del core IP com a caixa negra, així com el procés d'instal·lació de VUnit, el seu funcionament i els seus avantatges enfront d'altres tipus de verificacions. També es mostraran les etapes de verificació que s'han anat seguint fins a aconseguir verificació completa, des de la més senzilla fins a la més complexa.

També es detallarà la realització d'una interfície d'usuari per a poder executar la pròpia verificació d'una manera més intuïtiva i pràctica. A més de que arxius són els executables de la mateixa i el seu ús.

Abstract

This paper shows how to verify a RISC-V processor core IP in VUnit and partially support the RV32I instructions in its Single-Cycle version. It also shows the possible errors that can occur in the Multi-Cycle version if a bad segmentation is made or the errors are treated incorrectly.

The operation of the core IP as a black box will be detailed, as well as the installation process of VUnit, its operation and its advantages over other types of verifications. It will also show the verification stages that have been followed to achieve complete verification, from the simplest to the most complex.

It will also detail the creation of a user interface to be able to execute the verification itself in a more intuitive and practical way. It will also explain which files are executable and how to use them.

Índice general

Capítulo 1

Preámbulo

El punto de partida de este trabajo es el proyecto desarrollado por Álvaro Gomariz Pérez, Jesús Martinez García, Álvaro Villaescusa Tebar y Marc Sánchez Alfaro en la asignatura de Integración de Sistemas Digitales (ISDIGI), cursada durante el grado. Este trabajo está basado en el último proyecto de la asignatura ya mencionada, desarrollar un core RISC-V Single-Cycle y Multi-Cycle que daba soporte parcial y total (de forma voluntaria) para el ISA (conjunto de instrucciones) RV32I.

Los objetivos de este proyecto son:

- Realizar la verificación del RiscV como caja negra, únicamente conociendo lo que pasa en el exterior del propio procesador.
- Aprender a emplear VUnit en su totalidad.
- Realización de una interfaz gráfica para facilitar el uso de la verificación mediante PythonQT

Para poder realizar la verificación se va a emplear el proyecto ya mencionado anteriormente, este proyecto es funcional. En un proyecto real y completamente nuevo no va a ser funcional, entonces de todos los casos que se encuentran en un proyecto real:

- Diseño mal y verificación mal
- Diseño mal y verificación bien
- Diseño bien y verificación mal
- Diseño bien y verificación bien

Solo se podrán observar los dos últimos en la realización del proyecto. Para los futuros alumnos se moverán entre el segundo y el último caso.

A continuación se detallan las aplicaciones que se han empleado:

- QuestaSIM: Simulador empleado para la verificación.

- Visual Studio Code: Editor de código donde se van a editar archivos en System Verilog y en Python.
- Python versión 3.8.5: VUnit emplea Python, además de que es necesario para su instalación.

1.1. Tareas a desarrollar y plan de trabajo

Las tareas a seguir en el proyecto son las siguientes:

- Búsqueda de información sobre VUnit y RISC-V.
- Instalación de VUnit.
- Verificación sencilla mediante VUnit.
- Familiarizarse con el entorno de VUnit.
- Implementación del proyecto de ISDIGI.
- Verificación funcional de proyecto.
- Random Constraint Verification (RCV).
- Creación de un top genérico.
- Desarrollo de una interfaz de usuario.
- Utilizar otros procesadores.

En la siguiente imagen se muestra con detalle el flujo de trabajo dividido por semanas, dando comienzo en la segunda semana de Abril:

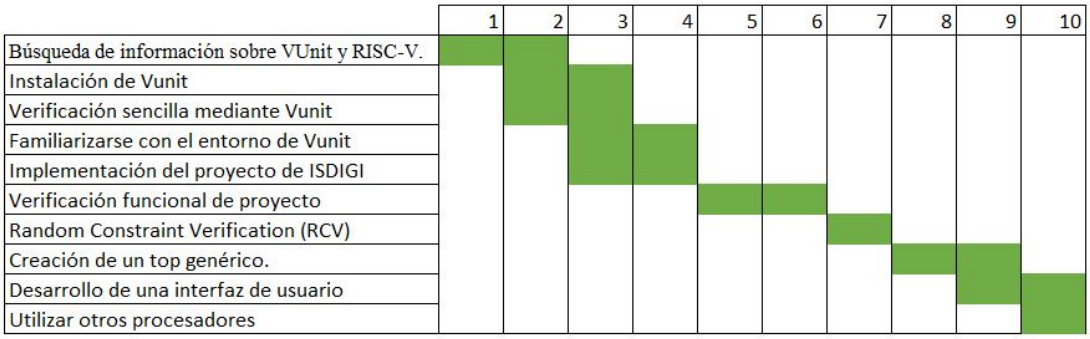

Figura 1.1: Flujo del proyecto por semanas

Capítulo 2

Introducción a RISC-V

Al tener que realizar una verificación de un core de un procesador basado en RISC-V hay que saber que es y como funcionan las instrucciones sin entrar en detalles de como realizar el core.

2.1. ¿Qué es RISC-V?

Es una arquitectura ISA de hardware libre donde el objetivo principal de sus autores es que se pueda emplear para cualquier propósito, incluso la realización y venta de chips basados en esta tecnología.

Como su propio nombre indica, RISC, el conjunto de instrucciones es reducido (Reduced Instruction Set Computer), esto hace que su uso sea para implementaciones rápidas, pequeñas y de bajo consumo y, que queden excluidos los proyectos que necesiten una arquitectura concreta o excesiva.

El proyecto RISC-V surgió en 2010 en la Universidad de California, Berkley, pero la gran mayoría de los trabajadores de esta tecnología son voluntarios o se encuentran en entornos distintos al de la Universidad.[1]

2.2. Convención de nombres en RISC-V

Para comprender como se desarrolla la propia convención de nombres lo mejor es entenderlo con el conjunto de instrucciones que se van a verificar:

- RV: Esto es la abreviación del propio Risc V. Todos los conjuntos de instrucciones de Risc V empiezan con estas siglas.
- 32: Indica el ancho del banco de registros en bits. En este caso el ancho del banco de registros es de 32 bits.
- I: Indica que soporta los números enteros.
- M: Indica que soporta la multiplicación y la división.

Observando este ejemplo se puede deducir que las instrucciones en Risc V se denotan de la siguiente manera:

- RV: Es común en todos los conjuntos de instrucciones
- Ancho de bits del banco de registros: Las opciones que soporta son 32, 64 y 128 bits
- Soportes y extensiones: Que es capaz de soportar el propio microprocesador

2.3. Conjunto de instrucciones

Para realizar la verificación hay que conocer que instrucciones hay que verificar. Pero, antes de observarlas hay que discriminar las instrucciones y entender la propia nomenclatura de las operaciones.

En cuanto a la discriminación de las instrucciones estas se pueden dividir en varios tipos dependiendo de como es la operación. Estas se discriminan por el propio OPCODE (Operation code, tipo de operación a realizar), estos pertenecen a los últimos 7 bits de la instrucción (del bit 6 al 0) y se pueden distinguir los siguientes grandes grupos:

- Instrucciones tipo I: Son aquellas que emplean un inmediato y una lectura de registro como operando.
- Instrucciones tipo R: Son aquellas que emplean las dos lecturas de registro como operando.
- Instrucciones tipo B: Son aquellas que realizan un salto si se cumplen las condiciones de la instrucción.
- Instrucciones tipo S: Son aquellas que realizan carga y descarga en la memoria de datos.
- Instrucciones tipo J: Son instrucciones de salto.

En la propia GreenCard del Risc V (donde se pueden observar todas las instrucciones de manera oficial) se detallan las instrucciones con la siguiente convención de nombres:

- R: Registro
- rd: Registro Destino
- rs1: Registro Salida 1
- rs2: Registro Salida 2
- Imm: Inmediato

Con toda esta información, ya se pueden detallar todas las instrucciones[2][3] a verificar a continuación:

Instrucción	Operación	Tipo
ADDI	R[rd] = R[rs1] + imm	I
SLTI	R[rd] = (R[rs1] <imm)? 1 : 0	I
SLTIU	R[rd] = (R[rs1] <imm)? 1 : 0	I
ANDI	R[rd] = R[rs1] & imm	I
ORI	R[rd] = R[rs1] | imm	I
XORI	R[rd] = R[rs1] $\wedge imm$	I
ADD	R[rd] = R[rs1] + R[rs2]	R
SUB	R[rd] = R[rs1] - R[rs2]	R
SLT	R[rd] = (R[rs1] <R[rs2])? 1 : 0	R
SLTU	R[rd] = (R[rs1] <R[rs2])? 1 : 0	R
AND	R[rd] = R[rs1] & R[rs2]	R
OR	R[rd] = R[rs1] | R[rs2]	R
XOR	R[rd] = R[rs1] $\wedge R[rs2]$	R
BEQ	if(R[rs1]==R[rs2]) PC = PC+imm	B
BNE	if(R[rs1]!=R[rs2]) PC = PC+imm	B
LW	R[rd]=32'bM[](31),M[[R[rs1]]+imm](31:0)	S
SW	M[R[rs1]+imm](31:0) = R[rs2](31:0)	S
LUI	R[rd]=32'b0,M[R[rs1]+imm](31:0)	U
AUIPC	R[rd]=PC+imm,12'b0	U
JAL	R[rd] = PC + 4; PC = PC + imm	UJ
JALR	R[rd] = PC + 4; PC = PC + R[s1]	I
BLT	if(R[rs1]<R[rs2]) PC = PC+imm	B
BLTU	if(R[rs1]<R[rs2]) PC = PC+imm	B
BGE	if(R[rs1]$\geq R[rs2])PC = PC + imm$	B
BGEU	if(R[rs1]$\geq R[rs2])PC = PC + imm$	B
SLL	R[rd] = R[rs1] $\ll R[rs2]$	R
SRL	R[rd] = R[rs1] $\gg R[rs2]$	R
SRA	R[rd] = R[rs1] $\gg R[rs2]$	R
SLLI	R[rd] = R[rs1] $\ll imm$	I
SRLI	R[rd] = R[rs1] $\gg imm$	I
SRAI	R[rd] = R[rs1] $\gg imm$	I

Tabla 2.1: Conjunto de instrucciones a verificar

Instrucción	[31-25]	[24-20]	[19-15]	[14-12]	[11-7]	[6-0]
ADD	0000000	rs2	rs1	000	rd	0110011
SUB	0100000	rs2	rs1	000	rd	0110011
AND	0000000	rs2	rs1	111	rd	0110011
SLL	0000000	rs2	rs1	001	rd	0110011
SLT	0000000	rs2	rs1	010	rd	0110011
SLTU	0000000	rs2	rs1	011	rd	0110011
OR	0000000	rs2	rs1	110	rd	0110011
XOR	0000000	rs2	rs1	100	rd	0110011
SRL	0000000	rs2	rs1	101	rd	0110011
SRA	0000000	rs2	rs1	101	rd	0110011
SW	0000000	rs2	rs1	101	rd	0110011

Tabla 2.2: Bits de las instrucciones R format y SW

Instrucción	[31-25]	[24-20]	[19-15]	[14-12]	[11-7]	[6-0]
ADDI	imm[11:5]	imm[4:0]	rs1	000	rd	0010011
SLTI	imm[11:5]	imm[4:0]	rs1	010	rd	0010011
SLTIU	imm[11:5]	imm[4:0]	rs1	011	rd	0010011
XORI	imm[11:5]	imm[4:0]	rs1	100	rd	0010011
ORI	imm[11:5]	imm[4:0]	rs1	110	rd	0010011
ANDI	imm[11:5]	imm[4:0]	rs1	111	rd	0010011
LW	imm[11:5]	imm[4:0]	rs1	010	rd	0000011

Tabla 2.3: Bits de algunas instrucciones I format y LW

Instrucción	[31-25]	[24-20]	[19-15]	[14-12]	[11-7]	[6-0]
SLLI	000000X	rs2	rs1	001	rd	0010011
SRLI	000000X	rs2	rs1	101	rd	0010011
SRAI	010000X	rs2	rs1	101	rd	0010011

Tabla 2.4: Bits de algunas instrucciones I format

Instrucción	[31-25]	[24-20]	[19-15]	[14-12]	[11-7]	[6-0]
BEQ	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
BNE	imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011
BLT	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011
BGE	imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011
BLTU	imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011
BGEU	imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011

Tabla 2.5: Bits de las instrucciones B format

Instrucción	[31-12]	[11-7]	[6-0]
LUI	imm[31:12]	rd	0110111
AUIPC	imm[31:12]	rd	0110111
JAL	imm[20|10:1|11|19:12]	rd	1101111

Tabla 2.6: Bits de las instrucciones U format y JAL

Instrucción	[31-25]	[24-20]	[19-15]	[14-12]	[11-7]	[6-0]
JALR	imm[11:5]	imm[4:0]	rs1	000	rd	1100111

Tabla 2.7: Bits de la instruccion JALR

2.4. Generación de inmediatos

Como se observa en el apartado anterior, los inmediatos se generan en base a la instrucción, pero, estos deben de ser de 32 bits. Para lograr alcanzar los 32 bits se realiza un desplazamiento del bit de mayor peso tal y como se observa a continuación:

1. La propia instrucción nos genera el siguiente inmediato: 1101 0001 1000 0110.
2. Duplicamos el bit de mayor peso y lo desplazamos hacia la izquierda hasta llegar a los 32 bits: 1111 1111 1111 1111 1101 0001 1000 0110.

Algunas instrucciones ya tienen internamente declarados los bits de mayor peso como puede ser el caso de AUIPC, en este caso se realiza un desplazamiento del bit de menor peso:

1. La propia instrucción nos genera el siguiente inmediato: 1101 0001 1000 0110.
2. Duplicamos el bit de mayor peso y lo desplazamos hacia la izquierda hasta llegar a los 32 bits: 1101 0001 1000 0110 0000 0000 0000 0000.

Capítulo 3

Introducción a VUnit

3.1. ¿Qué es VUnit?

VUnit[4] es una libreria de Python de código abierto (gratuita) que realiza tests continuos y automatizados para los lenguajes de VHDL, SystemVerilog (el empleado en el proyecto) y Verilog.

VUnit no reemplaza otros métodos tradicionales de verificación como puede ser UVM (Universal Verification Method), puede ser usado paralelamente con estos otros métodos para una mayor eficacia a la hora de verificar un diseño.

Además, VUnit no impone al verificador ningún módelo de verificación a seguir, sus tests pueden ser muy cortos, muy largos y contener los elementos que el propio verificador considere oportunos.

3.2. Ventajas de VUnit

VUnit fue diseñado para realizar pequeños tests automatizados de forma eficaz. En vez de utilizar muchos tests que tardan muchas horas en realizar una verificación (en un entorno laboral, estas verificaciones se realizan al finalizar la jornada laboral y se ejecutan a lo largo de la noche para comprobar a la mañana siguiente los errores) lo puedes realizar en unos pocos segundo (o minutos) con VUnit. Esto supone un gran ahorro para las empresas a la hora de realizar este tipo de tests. Esta es su principal ventaja.

El resto de ventajas se detallan a continuación:

- Crear unidades de test independientes.
- Capacidad de ejecutar tests en paralelo.
- Capacidad de generar una salida en JUnit para realizar una integración con Jenkins.

Como se puede observar, todas las ventajas son para el mismo proposito, agilizar las verificaciones de los diseños.

3.3. Instalación de VUnit

Previamente a instalar VUnit se requiere tener instalados los siguientes programas:

- Python, versión 3.8.5 o superior. Para emplear las propias librerias de VUnit.
- QuestaSIM. Para realizar las simulaciones y lanzar las verificaciones.

Una vez se ha revisado que se tienen estos programas en el ordenador se procede a su instalación desde la terminal (CMD):

```
pip install vunit_hdl
```

```
pip install -U vunit_hdl
```

El primer comando te instala VUnit, el segundo, lo actualiza (en caso de haber instalado una versión previa a la última).

Es probable que la instalación nos de el error que se muestra a continuación:

```
C:\Users\Antioxts>pip install -U vunit_hdl
Collecting vunit_hdl
  Using cached vunit_hdl-4.6.0.tar.gz (626 kB)
  Installing build dependencies ... done
  Getting requirements to build wheel ... done
  Installing backend dependencies ... done
    Preparing wheel metadata ... error
    ERROR: Command errored out with exit status 1:
     command: 'c:\users\antioxts\appdata\local\programs\python\python38-32\python.exe' 'c:\users\antioxts\appdata\local\programs\python\python38-32\lib\site-packages\pip\_vendor\pep517\_in_process.py' prepare_me
tadata_for_build_wheel 'C:\Users\ANTOXT~1\AppData\Local\Temp\tmpnzpum8im'
         cwd: C:\Users\Antioxts\AppData\Local\Temp\pip-install-k_yb2ywm\vunit-hdl
    Complete output (13 lines):
    running dist_info
    creating C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info
    writing C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info\PKG-INFO
    writing dependency_links to C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info\dependency_links.txt
    writing requirements to C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info\requires.txt
    writing top-level names to C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info\top_level.txt
    writing manifest file 'C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info\SOURCES.txt'
    reading manifest file 'C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info\SOURCES.txt'
    writing manifest file 'C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.egg-info\SOURCES.txt'
    creating 'C:\Users\Antioxts\AppData\Local\Temp\pip-modern-metadata-ujkicgx8\vunit_hdl.dist-info'
    Error in sitecustomize; set PYTHONVERBOSE for traceback:
    SyntaxError: (unicode error) 'utf-8' codec can't decode byte 0xed in position 0: unexpected end of data (sitecustomize.py, line 7)
    error: invalid command 'bdist_wheel'
    ----------------------------------------
ERROR: Command errored out with exit status 1: 'c:\users\antioxts\appdata\local\programs\python\python38-32\python.exe' 'c:\users\antioxts\appdata\local\programs\python\python38-32\lib\site-packages\pip\_vendor\
pep517\_in_process.py' prepare_metadata_for_build_wheel 'C:\Users\ANTOXT~1\AppData\Local\Temp\tmpnzpum8im' Check the logs for full command output.
WARNING: You are using pip version 20.1.1; however, version 22.0.2 is available.
You should consider upgrading via the 'c:\users\antioxts\appdata\local\programs\python\python38-32\python.exe -m pip install --upgrade pip' command.
```

Figura 3.1: Error en la instalación

Esto se soluciona introduciendo el siguiente comando en la CMD:

```
pip install wheel
```

Una vez realizado este comando, se procede a introducir los dos anteriores nuevamente.

En este punto VUnit está instalado pero no se ha configurado para que su simulador sea QuestaSIM, para ello se tienen que introducir dos variables de entorno la cual apunte al propio QuestaSIM tal y como muestra la siguiente imagen:

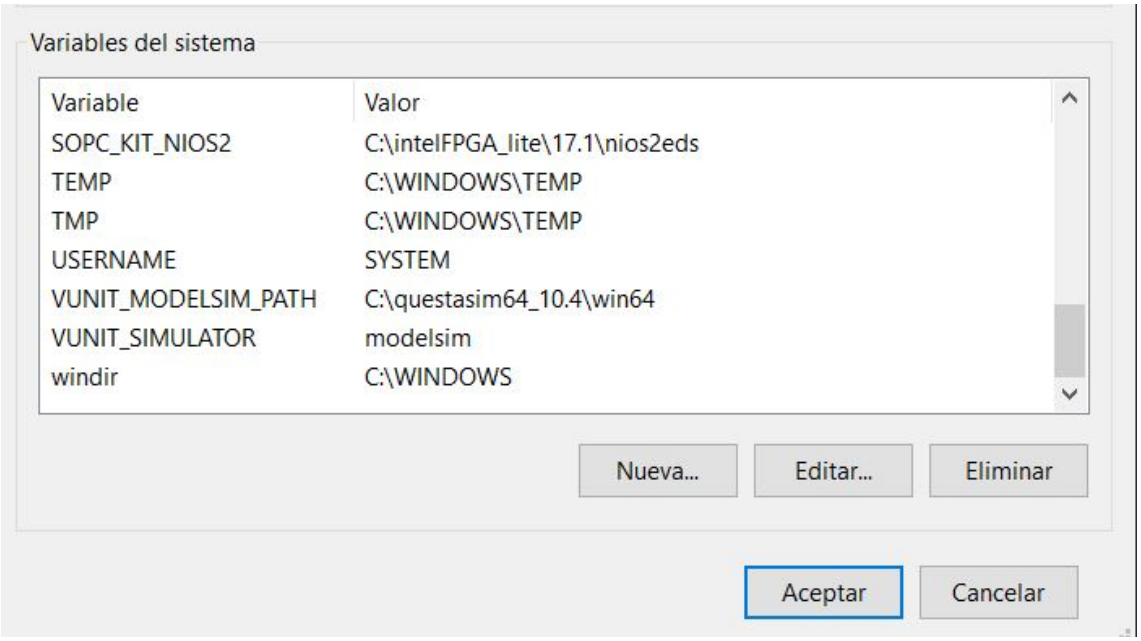

Figura 3.2: Variables de entorno

Para acceder a las variables de entorno tenemos que realizar lo siguiente:

1. Introducir 'Editar las variables de entorno del sistema' en el buscador de Windows.
2. Seleccionar la primera opción que aparece.
3. Seleccionar 'Variables de entorno'.
4. Seleccionar el segundo botón de 'Nueva'. Corresponde con el apartado de variables del sistema.
5. Introducirlas según la imagen anterior.
6. Reiniciar el ordenador (siempre que se modifica una variable de entorno hay que realizar un reinicio del sistema).

Para comprobar su instalación se realizará una verificación sencilla tal y como se detalla a continuación.

3.4. Verificación sencilla

La propia página de VUnit proporciona un propio fichero de la estructura de un test empleando SystemVerilog (comentando que se ha empleado y que no para realizar la verificación). A continuación se muestra dicha estructura y a explicar los elementos empleados:

```
`include "vunit_defines.svh"
```

```
module tb_example;
   `TEST_SUITE begin
      // Note: Do not place any code here (unless you are debugging
      // VUnit internals).

      `TEST_SUITE_SETUP begin
         // Your awesome common code
      end

      `TEST_CASE_SETUP begin
         //No empleado
      end

      `TEST_CASE("Your awesome name") begin
         //Your awesome code
      end

      `TEST_CASE("Your awesome name") begin
         //Your awesome code
      end

      `TEST_CASE_CLEANUP begin
         //No necesario en este caso
      end

      `TEST_SUITE_CLEANUP begin
        //No necesario en este caso
      end
   end;

   // The watchdog macro is optional, but recommended. If present, it
   // must not be placed inside any initial or always-block.
   `WATCHDOG(1ns);
endmodule
```

En el cual se observa lo siguiente:

- Los test case son los distintos tests que se van a realizar.
- El test suite setup sirve para tener un caso genérico predefinido ayudando a no escribir cantidades desmedidas de código.

Como se ha visto anteriormente, VUnit emplea python. A continuación se muestra un ejemplo de un fichero en Python para emplear VUnit.

```
from pathlib import Path
from vunit.verilog import VUnit

SRC_PATH = Path(__file__).parent / "src"

VU = VUnit.from_argv(compile_builtins=False)
VU.add_library("tb").add_source_files("*.sv")
VU.add_library("micro").add_source_files(SRC_PATH/"*.sv")

//Opciones de compilación aqui

VU.main()
```

En este código se observa como se apunta a los directorios correspondientes para empezar la verificación y la última línea nos indica que se va a realizar la ejecución del código de test (todavía no se ha indicado que tests tiene que realizar, únicamente indica que se va a ejecutar la verificación).

Así que copiando el código de la UART, disponible en la propia página web oficial de VUnit (se recomienda el uso de git para copiarlo en tu ordenador), se procede a introducir el siguiente comando en la CMD ejecutado desde el propio directorio donde se encuentra el archivo.

```
run.py
```

Este comando ejecuta VUnit y si todo está correcto debe de aparecer algo similar, tal y como en la siguiente imagen:

```
Starting tb.tb_top_duv.OR
Output file: C:\TFG\codigo_interfaz\vunit_out\test_output\tb.tb_top_duv.OR_538208e3a6980225511a0c42da75fbb7d6d9466c\output.txt
pass (P=1 S=0 F=0 T=1) tb.tb_top_duv.OR (3.1 seconds)

==== Summary ============================
pass tb.tb_top_duv.OR (3.1 seconds)
========================================
pass 1 of 1
========================================
Total time was 3.1 seconds
Elapsed time was 3.1 seconds
========================================
All passed!
```

Figura 3.3: Ejecución del primer test

Si aparece un error se recomienda revisar el apartado anterior por completo. En cambio, si se asemeja, ya se ha instalado VUnit correctamente en el dispositivo.

3.5. Instrucciones básicas de VUnit

En el apartado anterior se ha realizado una ejecución general de nuestro código de VUnit pasando por todos los tests correspondientes, pero, quizás esa información no nos sirve, o se quiere una información más detallada, incluso, abrir el propio simulador. Para ello, VUnit tiene varias opciones de ejecución. Aquí se expondran las más básicas:

```
run.py --compile
```

El comando anterior compila el código. Al lanzar la simulación también lo compila.

```
run.py --list
```

Hace una lista de todos los tests que hay actualmente en el fichero run.py. Este es muy útil para comprobar si faltan tests por realizar en el diseño sin tener que esperar a que se realice la simulación por completo para que aparezca el resumen.

```
run.py --dont-catch-exceptions
```

Si encuentra un error, continua simulando. Muy útil para comprobar el funcionamiento de los resets cuando el diseño está en una fase muy temprana y se requiere observar si han realizado bien el reset.

```
run.py --verbose
```

Para dar más información al usuario mostrando los mensajes que se van introduciendo en el propio test. Introduciendo la siguiente línea de código en unas líneas que se ejecutan constantemente se puede observar para que sirve el propio verbose:

```
$display("Este mensaje solo aparece en el Verbose");
```

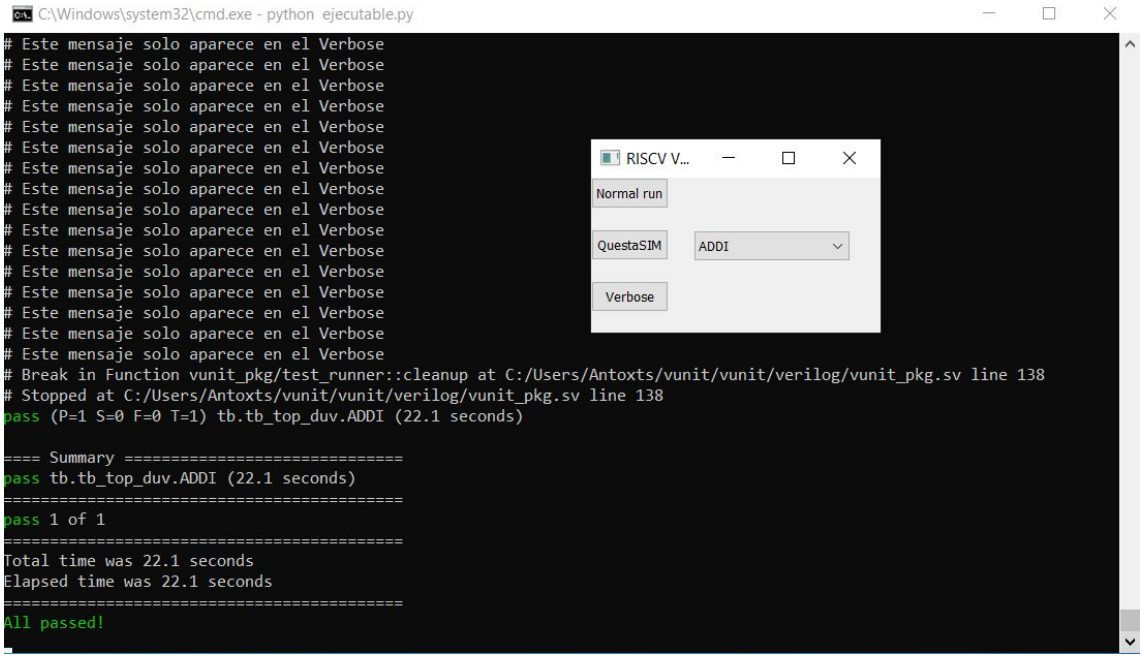

Figura 3.4: Ejemplo de uso de Verbose (empleando el verbose)

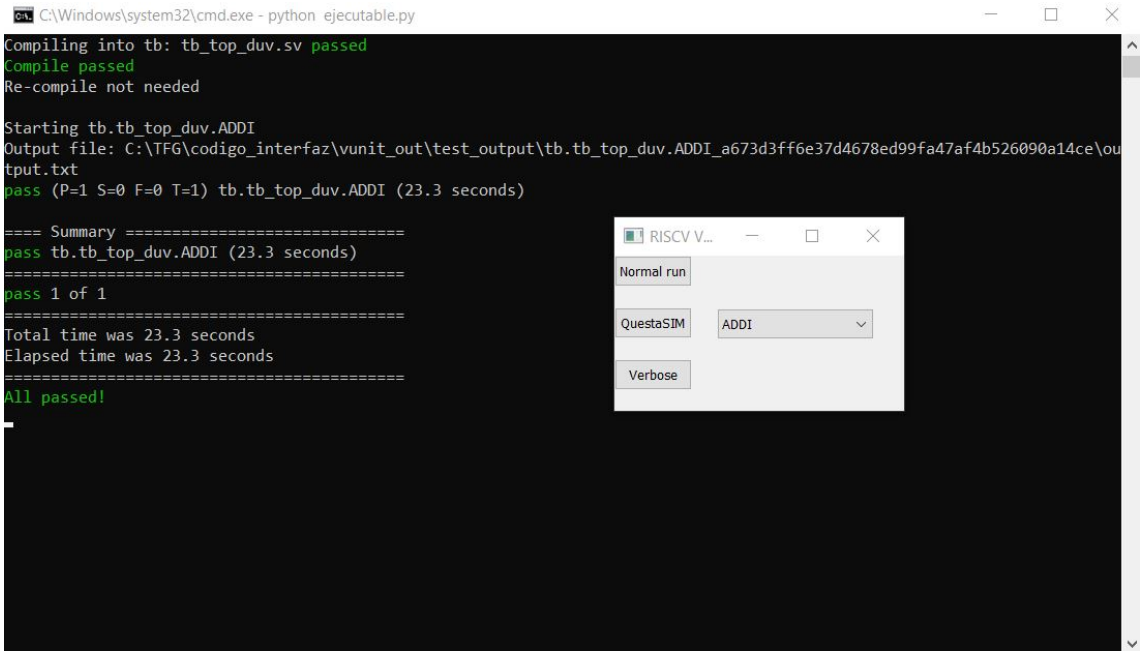

Figura 3.5: Ejemplo de uso de Verbose (sin emplearlo)

```
run.py --gui
```

Este es el más importante de todos. Abre el simulador que se está empleando en el momento, lo único malo es que hay que darle manualmente al botón de ejecutar en el propio simulador, pero es donde se obtiene la mayor información, visualizando las señales y que valores tienen en cada ciclo de reloj.

Cabe destacar que todas ellas se pueden combinar entre si generando más instrucciones, además se pueden ejecutar un test, varios tests, o todos ellos (este último es introducir solo run.py). A continuación se puede ver un ejemplo con los tests empleados en este proyecto:

```
run.py tb.tb_top_duv.AND
```

Este comando nos ejecuta únicamente el test llamado AND localizado en el fichero tb_top_duv de la librería tb.

```
run.py tb.tb_top_duv.AND tb.tb_top_duv.ADDI
```

Este comando nos ejecuta dos instrucciones, llamadas AND y ANDI.

Capítulo 4

Procesador a verificar

En este proyecto se va a trabajar desde el punto de vista de un ingeniero de verificación donde su función es verificar un diseño desconociendo como este ha sido realizado por dentro, únicamente con las especificaciones del mismo. Esto se consigue entablando un conversación mínima con un compañero de diseño y no observar como se ha realizado internamente el propio core del microprocesador (a pesar de estar trabajando con el mismo core).

El párrafo de arriba se puede comentar de otra manera, el ingeniero de verificación y el ingeniero de diseño van a obtener los mismos datos y las mismas especificaciones. Si las especificaciones se han transmitido claramente ambas partes no deberían de tener ningún problema en realizar el diseño y realizar la verificación.

4.1. Modelo caja negra de un Core RISC V

Como se ha comentado anteriormente, se han recibido las instrucciones de verificar un core de un microprocesador que tenga las siguientes características.

- Tiene que soportar todas las instrucciones RV32I.
- Su direccionamiento es de byte
- Siempre que se lea en el registro 0, hay que leer un 0, independientemente de lo que haya escrito.

Y a continuación se ha entregado a los ingenieros de verificación el siguiente modelo:

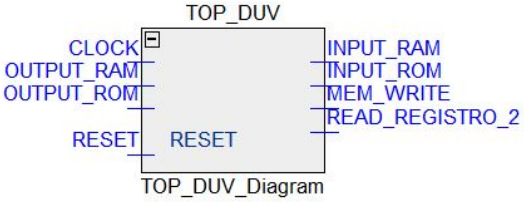

Figura 4.1: Instancia del core del microprocesador

De forma adjunta han entregado la memoria de instrucciones y la memoria de datos a las que se van a conectar.

4.1.1. Memoria de instrucciones

El objetivo de la memoria de instrucciones (ROM) es obtener la siguiente instrucción a realizar empleando uso del PC.

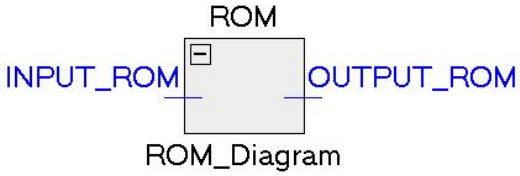

Figura 4.2: Instancia de la ROM

Cabe destacar que el direccionamiento es del RiscV es de byte, y el de la propia ROM es de palabra.

4.1.2. Memoria de datos

El objetivo de la memoria de datos (RAM) es trabajar con los datos que se están ejecutando actualmente.

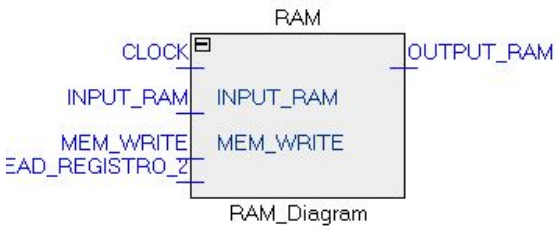

Figura 4.3: Instancia de la RAM

4.2. Verificación funcional

Tras obtener todas las especificaciones los ingenieros de verificación ya pueden empezar a realizar su verificación. Para ello, el primer objetivo es descubrir que realizar con las memorias de datos, de instrucciones y de registros

4.2.1. Memoria de datos y memoria de instrucciones

La memoria de instrucciones se decide desconectar e introducir instrucciones de manera aleatoria, respecto la memoria de la RAM se decide realizar un modelo ideal de la misma e instanciarla en el propio módulo de verificación, de esta manera cuando se tenga que realizar una escritura en la memoria de datos, nuestro modelo lo escribirá, además de que si se necesita realizar una lectura nuestro modelo la realizará

4.2.2. Memoria de registros

El primer problema que se plantea es verificar si una instrucción se ha realizado correctamente, por ejemplo, la instrucción add x1, x2, x3 realiza la suma de x2 más x3 y guarda el resultado en el registro x1. ¿Cómo se puede saber el valor de los registros x2 y x3? ¿Cómo se puede saber si se ha realizado correctamente el guardado del dato?

Como se observa esto es un gran problema a la hora de realizar la verificación, así que se puede optar por una única solución: realizar una salida extra (los valores de la memoria de registros) que no sea sintetizable para el diseño pero que si es compilable para una verificación. A continuación observamos como se declara, tanto en VUnit (como opción de compilación) como en SystemVerilog:

```
VU.set_compile_option("modelsim.vcom_flags", ["+cover=bs"])
VU.set_compile_option("modelsim.vlog_flags", ["+cover=bs","+define+
    no_sintetizable"])
```

En VUnit se declara como un define con el nombre de la variable (en este caso, no_sintetizable) precedido del símbolo de adición.

```
`ifdef no_sintetizable
//Your awesome code here
`endif
```

Esta declaración se ha de escribir en todas las partes de nuestro código que empleen un elemento no sintetizable para diseño pero si compilable en los de verificación. Esto nos permite poder realizar una lectura de la propia memoria de registros, así se puede conocer si la instrucción se realiza correctamente.

4.3. Random Constraint Verification

El Random Constraint Verification (RCV), como su propio nombre indica, es la generación aleatoria de señales con restricciones (constraints). Esto es empleado para realizar verificaciones de manera aleatoria, en nuestro caso, instrucciones que van a ser discriminadas por los valores fijos de cada instrucción.

4.3.1. Generación aleatoria

Para realizar la generación aleatoria de estímulos empleamos el siguiente esquema para cada instrucción:

```
class beq;
randc bit [31:0] instr;
    constraint instruccion {instr[14:12] == 3'b000;};
    constraint opcode {instr[6:0] == 7'b1100011;};
endclass
beq BEQ;
```

Se puede observar que empleamos la función randc que significa generar valores aleatorios distintos a los anteriores, de esta manera nos aseguramos verificar una instrucción distinta a las anteriores.

4.3.2. Cobertura

Tras haber generado las instrucciones de manera aleatoria, necesitamos saber si ha verificado todas las instrucciones además de si el diseño funciona como esperaba. Para comprobar esto tenemos la cobertura (coverage) y hay de dos tipos: cobertura funcional, este nos comprueba cuantas instrucciones se han ejecutado en porcentaje y el segundo nos muestra las líneas de código que se han ejecutado tanto en la verificación como en el diseño.

4.3.2.1. Cobertura funcional

Para poder comprender la cobertura funcional lo óptimo es exponer el código empleado:

```
covergroup beq_cv;
random1: coverpoint SENYALES.INSTRUCCION[31:15]{
    bins a[]={[0:$]};
}
instruccion: coverpoint SENYALES.INSTRUCCION[14:12] {
    bins b[]={3'b000};
}
random2: coverpoint SENYALES.INSTRUCCION[11:7] {
```

```
            bins c[]={[0:$]};
        }
        opcode: coverpoint SENYALES.INSTRUCCION[6:0] {
            bins d[]={7'b1100011};
        }
        random_cross: cross random1, random2;
        endgroup;
        beq_cv beq_cv_inst;
```

Este código hace referencia a la cobertura funcional de una instrucción donde se puede observar lo siguiente:

- covergroup es la declaración para realizar cobertura funcional.
- coverpoint hace referencia a que señal se va a comprobar por los valores que ha pasado.
- bins hace referencia cuantos valores son los que se tienen que comprobar para esa señal.
- cross junta varios coverpoints y se comprueba que todos los valores de un coverpoint se cumplan para todos los valores de los otros coverpoints.

4.3.2.2. Cobertura de código

La cobertura de código es mucho más sencilla ya que está integrada con el propio QuestaSIM, así que hay que indicarle a VUnit que habilite esta opción en el simulador:

```
VU.set_compile_option("modelsim.vcom_flags", ["+cover=bs"])
VU.set_compile_option("modelsim.vlog_flags", ["+cover=bs","+define+
    no_sintetizable"])
```

La cobertura de código se define con cover=bs.

Una vez habilitado se puede observar que líneas de código se han ejecutado y cuales no en el propio QuestaSIM. Tal y como se muestra a continuación:

```
1303        end
1304    `TEST_CASE("ADD") begin
1305            ADD = new();
1306            tCLOCK=1'b0;
1307            count = 0;
1308            add_cv_inst = new;
1309            @(posedge tCLOCK);
1310            fd = $fopen({duv_TOP_DUV.PATH,"/output_file/instrucciones/add.txt"}, "w");
1311            fhazard = $fopen({duv_TOP_DUV.PATH,"/output_file/hazard/add_hazard.txt"}, "w");
1312            while (add_cv_inst.get_coverage()<RCV)
1313            begin
1314                ADD_task();
1315                if (count != 0)
1316                begin
1317                    if (registro_escritura_anterior == SENYALES.INSTRUCCION[11:7])
1318                        begin
1319
1320                            $fwriteh(fhazard, instruccion_anterior);
1321                            $fwriteh(fhazard, "\n");
1322                            $fwriteh(fhazard, "Hazard tipo WAW en la linea:  %d", count);
1323                            $fwriteh(fhazard, "\n");
1324                            $fwriteh(fhazard, SENYALES.INSTRUCCION);
1325                            $fwriteh(fhazard, "\n");
1326                            $fwriteh(fhazard, "\n");
1327
1328                        end
1329
1330                    if (registro_lectura_anterior_1 == SENYALES.INSTRUCCION[11:7])
1331                        begin
1332
1333                            $fwriteh(fhazard, instruccion_anterior);
1334                            $fwriteh(fhazard, "\n");
1335                            $fwriteh(fhazard, "Hazard tipo WAR en la linea:  %d", count);
1336                            $fwriteh(fhazard, "\n");
1337                            $fwriteh(fhazard, SENYALES.INSTRUCCION);
1338                            $fwriteh(fhazard, "\n");
1339                            $fwriteh(fhazard, "\n");
```

Figura 4.4: Code coverage sin ejecutar las líneas de código

```
1303        end
1304    `TEST_CASE("ADD") begin
1305            ADD = new();
1306            tCLOCK=1'b0;
1307            count = 0;
1308            add_cv_inst = new;
1309            @(posedge tCLOCK);
1310            fd = $fopen({duv_TOP_DUV.PATH,"/output_file/instrucciones/add.txt"}, "w");
1311            fhazard = $fopen({duv_TOP_DUV.PATH,"/output_file/hazard/add_hazard.txt"}, "w");
1312            while (add_cv_inst.get_coverage()<RCV)
1313            begin
1314                ADD_task();
1315                if (count != 0)
1316                begin
1317                    if (registro_escritura_anterior == SENYALES.INSTRUCCION[11:7])
1318                        begin
1319
1320                            $fwriteh(fhazard, instruccion_anterior);
1321                            $fwriteh(fhazard, "\n");
1322                            $fwriteh(fhazard, "Hazard tipo WAW en la linea:  %d", count);
1323                            $fwriteh(fhazard, "\n");
1324                            $fwriteh(fhazard, SENYALES.INSTRUCCION);
1325                            $fwriteh(fhazard, "\n");
1326                            $fwriteh(fhazard, "\n");
1327
1328                        end
1329
1330                    if (registro_lectura_anterior_1 == SENYALES.INSTRUCCION[11:7])
1331                        begin
1332
1333                            $fwriteh(fhazard, instruccion_anterior);
1334                            $fwriteh(fhazard, "\n");
1335                            $fwriteh(fhazard, "Hazard tipo WAR en la linea:  %d", count);
1336                            $fwriteh(fhazard, "\n");
1337                            $fwriteh(fhazard, SENYALES.INSTRUCCION);
1338                            $fwriteh(fhazard, "\n");
1339                            $fwriteh(fhazard, "\n");
1340
1341                        end
```

Figura 4.5: Code coverage ejecutando algunas líneas de código

En la primera imagen se observa todo con cruces, esto significa que esas líneas de código no han sido ejecutadas, esto es correcto debido a que en ese instante se estaba comprobando otra instrucción, en cambio, en la segunda ha sido un test muy rápido y se observa que algunas se han ejecutado y otras no. Esto también es correcto.

4.4. Creación del test setup

Como se ha comentado anteriormente cada instrucción es independiente, esto genera que al realizar el test correspondiente a las instrucciones ADDI la memoria de registros quede intacta en el siguiente test. Esto ocasiona que los tests que se están realizando den como resultado un test fallido o un test verificado y sea erróneo.

Para ello VUnit nos presenta el test setup

4.4.1. ¿Que es un test setup?

A la hora de realizar muchos tests que tengan una configuración inicial predefinida se emplea el test setup, esto habilita que las líneas de código introducidas en el test setup se ejecuten previamente al inicio del test. Esto ocasiona limpieza del código de verificación y que los errores de la configuración inicial estén en las líneas del test setup.

4.4.2. Declaración del test setup

El test setup otorga una configuración inicial de los registros, por tanto, se procede a escribir la instrucción ADDI para tener una configuración inicial en la propia memoria de registros. El código del test setup queda de la siguiente manera

```
`TEST_SUITE_SETUP begin
       // Here you will typically place things that are common to
       // all tests, such as asserting the reset signal and starting
       // the clock(s).
       $display("Running test suite setup code");
       RELLENAR = new();
       tCLOCK=1'b0;
       tRESET=1'b0;
       count = 0;

       $display("Reseteamos el procesador");
       @(posedge tCLOCK);
       tRESET=1'b1;
       while (count < RELLENO)
       begin
           RELLENAR.random();
           SENYALES.INSTRUCCION = RELLENAR.instr;
           aux={{20{SENYALES.INSTRUCCION[31]}},SENYALES.INSTRUCCION[31:20]} +
   MEM[SENYALES.INSTRUCCION[19:15]];
           @(posedge tCLOCK);
           #1ns;
            count = count + 1;
       end

    end
```

Como se observa, esta vez se está llamando a la función ADDI, sino a la función rellenar, esta no emplea randc ya que no son necesarios para una cobertura, solo interesa tener valores en la propia memoria de registros y el código correspondiente a esta función es:

```
        class rellenar;
            bit [31:0] instr;
                function void random;
                    instr[31:15] = $random();
                    instr[14:12] = 3'b000;
                    instr[11:7] = $random();
                    instr[6:0] = 7'b0010011;

                endfunction
        endclass
        rellenar RELLENAR;
```

Cabe recalcar que el test setup no realiza ninguna comprobación, debido a que un diseño puede estar mal, y si esto es así, el usuario no podrá realizar nada, para ello se han deshabilitado todos los checks correspondientes.

4.5. Realización de los distintos test

Dependiendo del tipo de instrucción, el test puede ir variando un poco en la estructura, así que se mostraran los distintos tipos de test realizados.

4.5.1. Test general

Este es el test más sencillo ya que sigue la siguiente estructura:

1. Llamamos a los constructores de los covergroups y la generación aleatoria.
2. Realizamos la operación correspondiente y la guardamos en la variable correspondiente.
3. En el siguiente ciclo de reloj se comprueba el registro de destino con el valor de la variable.
4. Actualizamos el valor correspondiente al coverage.

A continuación se puede observar con el código correspondiente:

```
SRAI = new();
srai_cv_inst = new;
tCLOCK=1'b0;
@(posedge tCLOCK);
while (srai_cv_inst.get_coverage()<RCV)
    begin
    assert (SRAI.randomize()) else    $fatal("Falla randomizacion en SRAI");
    SENYALES.INSTRUCCION = SRAI.instr;
    if(SENYALES.INSTRUCCION[19:15] == 0)
        begin
            `CHECK_EQUAL(MEM[0], 0, "Registro 0 tiene que leer un 0");
        end
    aux = $signed(MEM[SENYALES.INSTRUCCION[19:15]]) >>> {{20{SENYALES.
    INSTRUCCION[31]}},SENYALES.INSTRUCCION[31:20]}[4:0];
    @(posedge tCLOCK);
    #1ns;
    if(SENYALES.INSTRUCCION[11:7] != 0)
```

```
        begin
            `CHECK_EQUAL(MEM[SENYALES.INSTRUCCION[11:7]], aux,"Falla en SRAI");
        end

    srai_cv_inst.sample();


    end
```

Como se puede observar, hay dos bloques ifs con dos asserts, estos bloques corresponden a la lectura y escritura en el registro 0. Si se lee en el registro 0, hay que leer un 0. Si se escribe en el registro 0, es indiferente si ha escrito o no, solo es importante la lectura.

4.5.2. Instrucciones con salto

Las instrucciones con salto son distintas, ya que pueden realizar un salto que modifican el propio PC del microprocesador. Para ello la estructura va a ser la misma pero con un PC interno en el propio microprocesador:

```
always @(posedge tCLOCK, negedge tRESET)
    if (!tRESET)
                                                    begin
                                                                    PC<=0;
                                                    end
            else
        begin
            if((SENYALES.INSTRUCCION [6:0] == 7'b1100011)||(SENYALES.
    INSTRUCCION [6:0] == 7'b1101111)||(SENYALES.INSTRUCCION [6:0] == 7'b1100111
    )) begin

            end else begin
                PC <= PC+4;
            end
        end
```

Este PC es muy sencillo, siempre se le va a sumar 4 a no ser que sea una instrucción que pueda conllevar un salto, el cual, es recalculado dentro del propio test de la instrucción:

```
BLTU = new();
bltu_cv_inst = new;
assert (BLTU.randomize()) else    $fatal("Falla randomizacion en BLTU");
SENYALES.INSTRUCCION = BLTU.instr;
if((SENYALES.INSTRUCCION[19:15] == 0) || (SENYALES.INSTRUCCION[24:20] == 0))
    begin
        `CHECK_EQUAL(MEM[0], 0, "Registro 0 tiene que leer un 0");
    end
    #1ns;
    if ($unsigned(MEM[SENYALES.INSTRUCCION[19:15]]) < $unsigned(MEM[SENYALES.
    INSTRUCCION[24:20]]))
    begin
        PC = PC + {{18{SENYALES.INSTRUCCION[31]}},SENYALES.INSTRUCCION[31],
    SENYALES.INSTRUCCION[7],SENYALES.INSTRUCCION[30:25],SENYALES.INSTRUCCION
    [11:8],1'b0};
    end else begin
        PC = PC + 4;
    end
```

```
    aux2 = PC [11:2];
    @(posedge tCLOCK);
    #1ns;
    `CHECK_EQUAL(SENYALES.PC, aux2);
```

Por último realiza la comprobación de si ha realizado el salto correspondiente o no ha realizado ningún tipo de salto, tal y como se muestra en el código anterior.

4.6. Posibles pérdidas en la segmentación

Este microprocesador ejecuta la instrucción en un solo ciclo, pero, se ha pedido realizar un estudio de las instrucciones que puedan ocasionar problemas a la hora de realizar una segmentación para que los ingenieros de diseño sepan donde focalizar los problemas a la hora de realizar la segmentación del microprocesador se ha decidido generar dos archivos informativos para cada instrucción. Uno que contenga los errores y que dos instrucciones los generan y otro que contenga las instrucciones a realizar.

4.6.1. ¿Qué es la segmentación?

Realizar una segmentación es dividir la tarea en pequeñas tareas que se pueden hacer simultáneamente. En el caso del microprocesador se pueden dividir en cinco etapas:

1. IF: Obtención de la instrucción a realizar.
2. ID: Búsqueda en la memoria de registros.
3. EX: Ejecución de la instrucción.
4. MEM: Búsqueda de operandos en memoria.
5. WB: Almacenamiento de resultados.

Todos estas etapas se pueden realizar simultáneamente, pero, ¿segmentar tiene una mejora en las características del microprocesador? La respuesta es un si, en el microprocesador sin segmentar, para realizar la instrucción más crítica (aquella que limita la frecuencia) puede tardar un tiempo elevado, en cambio, si dividimos esta instrucción en etapas la frecuencia de trabajo aumenta, por ello si se realiza en un conjunto de instrucciones el microprocesador segmentado va a realizar más operaciones, debido a que se ha aumentado la frecuencia de trabajo.

Al realizar la segmentación es posible que algunos datos no se procesen correctamente o incluso que algunas instrucciones no se ejecuten correctamente del todo. Todo esto son las pérdidas que se pueden ocasionar en un microprocesador, así que se evaluarán los riesgos que se pueden ocasionar y acarrear pérdidas en el mismo.

4.6.2. Tipos de pérdidas

Para poder generar los ficheros de pérdidas hay conocer los distintos tipos de pérdidas que se pueden ocasionar al realizar una segmentación del microprocesador e introducir las instrucciones.

Es posible que se produzcan más tipos de pérdidas, pero estas depende de como se haya decidido realizar la segmentación del propio microprocesador.

4.6.2.1. Riesgos de datos

Estos riesgos se producen cuando se genera una dependencia entre la instrucción actual y la anterior. La instrucción actual necesita los datos de la instrucción anterior. Hay de tres tipos:

- Tipo WAR (Write After Read): Se produce cuando la instrucción actual ha almacenado el dato sin ser leido por la instrucción anterior.

```
add  x1 , x2 , x3
add  x2 , x5 , x6
```

- Tipo WAW (Write After Write): Se produce cuando la instrucción actual almacena el dato antes de ser almacenado por la instrucción anterior.

```
add  x1 , x2 , x3
add  x1 , x5 , x6
```

- Tipo RAW (Read After Write): Se produce cuando se intenta leer un dato que aún no ha sido almacenado.

```
add  x1 , x2 , x3
add  x2 , x1 , x6
```

4.6.2.2. Riesgos de control

Este tipo de riesgo sucede cuando hay una instrucción de salto, debido a que es necesario matar varios procesos de ejecución para poder reanudar con la propia funcionalidad del microprocesador.

4.6.3. Generación de ficheros

Una vez conocemos los tipos de pérdidas se tienen que generar los ficheros, primero se empieza con el fichero de instrucciones, el cual va escribiendo instrucciones en un fichero txt. A continuación se muestra el resultado de un fichero de instrucciones:

and: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
006e79b3
00b6fab3
003f70b3
0117f333
01687e33
01b0fcb3
013977b3
00d1f533
008c7b33
0034f9b3
00fd7233
0055f0b3
0162f333
018b7d33
0133fe33
015277b3
0086f533
010f78b3
00f7fb33
00a87233
0000f0b3
00c97fb3
01f1fd33
01ac7433
0104f7b3
01cd7a33
0175f8b3
00a2fb33
002b7133
00c3f233
00927fb3
01a6fd33
00ef76b3
01c7f433
01967a33
0120f8b3
01e973b3
0041f133
00107c33
00e4ffb3
006d75b3
00b5f6b3
```

Figura 4.6: Ejemplo de archivo de salida con instrucciones

Una vez generado el fichero de instrucciones también tenemos que introducir la estructura de un fichero de pérdidas el cual se visualiza para una instrucción de la siguiente forma:

```
f38b8f13
Hazard tipo WAR en la linea:        64
3afd8b93
```

Se puede observar que se muestra la instrucción anterior, la instrucción actual y que instrucción del fichero de instrucciones la ha generado. Por último hay que observar la propia salida del fichero txt si es correcta:

addi_hazard: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
d3a58293
Hazard tipo WAR en la linea:         302
dac00593

145e0313
Hazard tipo RAW en la linea:         312
9d830493

e68a8c93
Hazard tipo RAW en la linea:         335
ef3c8813

429d0293
Hazard tipo WAR en la linea:         345
22688d13

54928b93
Hazard tipo RAW en la linea:         356
5d2b8c13

16e70a93
Hazard tipo RAW en la linea:         360
463a8113

9f7b0f93
Hazard tipo WAR en la linea:         369
92250b13

fbf90e93
Hazard tipo RAW en la linea:         380
8b2e8513

5fb48313
Hazard tipo WAR en la linea:         412
d2d18493

1b478213
Hazard tipo RAW en la linea:         422
15920513
```

Figura 4.7: Ejemplo de figura

Se ha optado por realizar solo para las pérdidas en base a los datos, ya que las pérdidas ocasionadas por un salto siempre van a existir. Además, es muy complicado generar una secuencia de instrucciones finita con instrucciones que incluyan saltos.

Por último, hay que observar la lógica para escribir en los ficheros, para ello se ha optado por guardar en las correspondientes variables la dirección de la memoria de registros e ir comparándolas para realizar la escritura en los ficheros. Finalmente la verificación de una sola instrucción tiene el siguiente formato:

```
`TEST_CASE("ADD") begin
            ADD = new();
            tCLOCK=1'b0;
            count = 0;
            add_cv_inst = new;
            @(posedge tCLOCK);
            fd = $fopen({duv_TOP_DUV.PATH,"/output_file/instrucciones/add.txt"
   }, "w");
            fhazard = $fopen({duv_TOP_DUV.PATH,"/output_file/hazard/add_hazard.
   txt"}, "w");
            while (add_cv_inst.get_coverage()<RCV)
            begin
                ADD_task();
                if (count != 0)
                begin
                    if (registro_escritura_anterior == SENYALES.INSTRUCCION
```

```
[11:7])
                    begin

                        $fwriteh(fhazard, instruccion_anterior);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "Hazard tipo WAW en la linea:  %d
", count);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, SENYALES.INSTRUCCION);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "\n");

                    end

                if (registro_lectura_anterior_1 == SENYALES.INSTRUCCION
[11:7])
                    begin

                        $fwriteh(fhazard, instruccion_anterior);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "Hazard tipo WAR en la linea:  %d
", count);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, SENYALES.INSTRUCCION);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "\n");

                    end

                if (registro_lectura_anterior_2 == SENYALES.INSTRUCCION
[11:7])
                    begin

                        $fwriteh(fhazard, instruccion_anterior);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "Hazard tipo WAR en la linea:  %d
", count);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, SENYALES.INSTRUCCION);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "\n");

                    end

                if (registro_escritura_anterior == SENYALES.INSTRUCCION
[19:15])
                    begin

                        $fwriteh(fhazard, instruccion_anterior);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "Hazard tipo RAW en la linea:  %d
", count);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, SENYALES.INSTRUCCION);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "\n");

                    end
```

```
                if (registro_escritura_anterior == SENYALES.INSTRUCCION
[24:20])
                    begin

                        $fwriteh(fhazard, instruccion_anterior);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "Hazard tipo RAW en la linea:  %d
", count);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, SENYALES.INSTRUCCION);
                        $fwriteh(fhazard, "\n");
                        $fwriteh(fhazard, "\n");

                    end
            end
                count = count + 1;
                $fwriteh(fd, SENYALES.INSTRUCCION);
                $fwriteh(fd, "\n");
                registro_escritura_anterior = SENYALES.INSTRUCCION[11:7];
                registro_lectura_anterior_1 = SENYALES.INSTRUCCION[19:15];
                registro_lectura_anterior_1 = SENYALES.INSTRUCCION[24:20];
                instruccion_anterior = SENYALES.INSTRUCCION;
                add_cv_inst.sample();


        end
        $fclose(fd);
        $fclose(fhazard);

    end
```

Como se observa, al introducir esta lógica se ha extendido en gran medida el código para la verificación de una sola instrucción

4.7. Generación de un test con todas las instrucciones

Finalmente se decidió realizar un test enfocado a la realización de instrucciones simultáneas. Como se ha explicado anteriormente, las instrucciones de salto son falsas, si se siguiesen todas las instrucciones con un orden secuencial y estrictamente, probablemente esta ejecución en un entorno real tendría bucles infinitos, pero, aquí se va a comprobar la funcionalidad de introducir instrucciones una a una sin tener el conocimiento de cual va a ser la siguiente.

Para realizar este test se ha optado porque todas las instrucciones tengan el mismo porcentaje de aparición, y, que con una probabilidad muy baja del 0.0001 se ejecutase un reset, cuando se realice el reset 10 veces se podrá dar por concluida la verificación. El código de esta verificación se muestra a continuación:

```
    `TEST_CASE("TODAS_LAS_INSTRUCCIONES")
      begin
        ADDI = new();
        SLTI = new();
        SLTIU = new();
        ANDI = new();
```

```
        ORI = new();
        XORI = new();
        ADD = new();
        SUB = new();
        SLT = new();
        SLTU = new();
        AND = new();
        OR = new();
        XOR = new();
        BEQ = new();
        BNE = new();
        LW = new();
        SW = new();
        LUI = new();
        AUIPC = new();
        JAL = new();
        JALR = new();
        BLT = new();
        BLTU = new();
        BGE = new();
        BGEU = new();
        SLL = new();
        SRL = new();
        SRA = new();
        SLLI = new();
        SRLI = new();
        SRAI = new();
        fd = $fopen({duv_TOP_DUV.PATH,"/output_file/instrucciones/todas_juntas.
    txt"}, "w");

        tCLOCK = 1'b0;
        numero_resets = 0;
        RESET_task();
        while(numero_resets <= 100)
        begin
            PORCENTAJE_INSTRUCCION = $urandom_range(0,309999);
            if((0<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION < 30))
                begin
                    RESET_task();
                    numero_resets++;
                end
            if((30<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION)&&(
    PORCENTAJE_INSTRUCCION<=10029))
                begin
                    ADDI_task();
                end

            if((10029<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
    <=20028))
                begin
                    SLTI_task();
                end

            if((20028<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
    <=30027))
                begin
                    SLTIU_task();
                end
```

```
        if((30027<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=40026))
            begin
                ANDI_task();
            end

        if((40026<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=50025))
            begin
                ORI_task();
            end

        if((50025<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=60024))
            begin
                XORI_task();
            end

        if((60024<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=70023))
            begin
                ADD_task();
            end

        if((70023<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=80022))
            begin
                SUB_task();
            end

        if((80022<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=90021))
            begin
                SLT_task();
            end

        if((90021<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=100020))
            begin
                SLTU_task();
            end

        if((100020<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=110019))
            begin
                AND_task();
            end

        if((110019<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=120018))
            begin
                OR_task();
            end

        if((120018<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=130017))
            begin
```

```
                    XOR_task();
                end

            if((130017<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=140016))
                begin

                    BEQ_task();
                end

            if((140016<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=150015))
                begin

                    BNE_task();
                end

            if((150015<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=160014))
                begin
                    LW_task();
                end

            if((160014<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=170013))
                begin

                    SW_task();
                end

            if((170013<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=180012))
                begin
                    LUI_task();
                end

            if((180012<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=190011))
                begin
                    AUIPC_task();
                end

            if((190011<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=200010))
                begin

                    JAL_task();
                end

            if((200010<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=210009))
                begin

                    JALR_task();
                end

            if((210009<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=220008))
```

```
            begin

                BLT_task();
            end

        if((220008<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=230007))
            begin

                BLTU_task();
            end

        if((230007<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=240006))
            begin

                BGE_task();
            end

        if((240006<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=250005))
            begin

                BGEU_task();
            end

        if((250005<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=260004))
            begin
                SLL_task();
            end

        if((260004<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=270003))
            begin
                SRL_task();
            end

        if((270003<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=280002))
            begin
                SRA_task();
            end

        if((280002<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=290001))
            begin
                SLLI_task();
            end

        if((290001<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=300000))
            begin
                SRLI_task();
            end

        if((290001<=PORCENTAJE_INSTRUCCION) && (PORCENTAJE_INSTRUCCION
<=309999))
```

```
                begin
                    SRAI_task();
                end
                $fwriteh(fd, SENYALES.INSTRUCCION);
                $fwriteh(fd, "\n");
        end
        $fclose(fd);
    end
```

4.8. Creación de una interfaz

En este momento, si el usuario decidía abrir el propio QuestaSIM para obtener una mayor información de que sucede en su propio diseño no podía visualizar que sucedía mediante señales. Esto es debido a que el propio VUnit realiza una simulación optimizada, sin generar las señales de entrada y salida.

La solución más sencilla es generar una interfaz. La interfaz sirve para encapsular todas las señales en un bloque, realizando las interconexiones entre los distintos componentes de una verificación más sencillos, además de que permite al propio QuestaSIM obtener las señales de entrada y salida. A continuación se muestra el código de la interfaz:

```
`timescale 1ns/1ps
interface test_if (  input  bit  CLOCK,  input  bit  RESET) ;


  parameter mem_depth=1024;
  parameter size=32;
  logic MEM_WRITE;
  logic [$clog2(mem_depth-1)-1:0] INPUT_RAM;
  logic  [$clog2(mem_depth-1)-1:0] PC; //INPUT_ROM
  logic [size-1:0] READ_REGISTRO_2;
  logic [size-1:0] OUTPUT_RAM;
  logic [size-1:0] INSTRUCCION; //OUTPUT_ROM


endinterface
```

Para observar las señales se tiene que realizar lo siguiente:

1. Seleccionar el elemento ”Señales”.
2. Seleccionar la señal a visualizar.
3. Arrastrarla al visualizador de señales.
4. Inicializar la simulación con QuestaSIM

A continuación se observa de manera más gráfica.

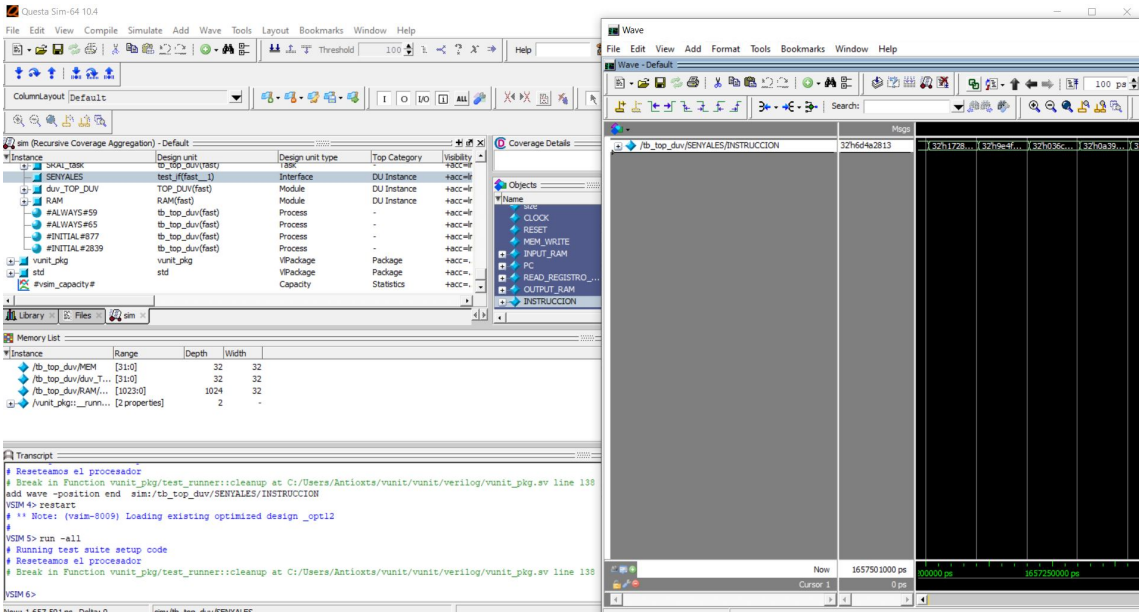

Figura 4.8: Ejemplo de como mostrar formas de onda

4.9. Creación de un fichero top genérico

Una vez se ha realizado toda la funcionalidad de la verificación hay que generar un archivo top con las instrucciones correspondientes a realizar para introducir distintos microprocesadores y así poder verificarlo:

```
module TOP_DUV (CLOCK, RESET, INPUT_RAM, OUTPUT_RAM, INPUT_ROM, OUTPUT_ROM,
    MEM_WRITE, READ_REGISTRO_2,
                `ifdef no_sintetizable
                                mem_registros
                                `endif);

                string PATH = "C:/YourAwesomeDirectory/ItsAwesome"; //Indica el
     path donde vas a guardar la carpeta

                parameter size = 32;
                parameter mem_depth = 1024;

                input CLOCK, RESET;

                output logic [$clog2(mem_depth-1)-1:0] INPUT_RAM,INPUT_ROM; //
    Entradas a la ROM y a la RAM

                output logic    MEM_WRITE; //Indica si hay que escribir en la
    RAM

                output logic [size-1:0] READ_REGISTRO_2; //Lectura
    correspondiente al segundo registro

                input logic [size-1:0] OUTPUT_RAM,OUTPUT_ROM;   //Salidas de la
     ROM (instrucciones) y de la RAM
```

```
                `ifdef no_sintetizable
                                output logic [size-1:0] mem_registros [size
    -1:0]; //Memoria de registros, declarar con el ifdef
                `endif


                //INTRODUCIR AQUÍ LA DECLARACIÓN DE VUESTRO PROPIO CORE Y
    OPERACIONES CORRESPONDIENTES


                //FINALIZAR CON LA ASIGNACIÓN DE LA MEMORIA DE REGISTROS


endmodule
```

Se puede observar perfectamente que tiene las instrucciones necesarias para que todo el mundo pueda introducir sus microprocesadores sin ningún tipo de problema.

4.10. Verificación de un microprocesador

Una vez se ha acabado de realizar todo el proyecto se decidió realizar una verificación para observar si había errores en el microprocesador realizado en la asignatura de integración de sistemas digitales. Este microprocesador presentó seis errores, un número bastante elevado:

- No funcionaba el reset: Esto se soluciona introduciendo la señal de reset en los registros, ya que este no estaba declarado.
- El registro cero no leía siempre un cero: Para solucionar esto, se ha optado por prohibir escribir en el registro cero en los propios registros.
- Fallo en las instrucciones SLTI, SLTIU, SRA y SRAI. Este fallo se encontró visualizando las formas de onda, estas estaban mal porque siempre generaban un 0. Se decidió revisar los operandos empleados tanto en la instrucción como en el test

Tras haber corregido los errores del microprocesador y los errores en las propias operaciones del test y observado las formas de onda nuevamente en las instrucciones, todos es correcto, por lo tanto se puede decir que este microprocesador es funcional.

Ahora la pregunta es la siguiente: ¿Se puede dar por concluido que el core del microprocesador es 100 % funcional si consigue pasar el test? La respuesta es un no. Es posible que bajo ciertas condiciones muy concretas el microprocesador pueda fallar, pero, el objetivo de este test es hacerlo funcional en la gran mayoría de casos y que los fallos bajo unas condiciones muy concretas sucedan muy pocas veces.

Otra pregunta a realizar es la siguiente: ¿la verificación es fiable? Para ello se va a realizar un estudio a continuación mostrando las visualizaciones de onda y realizando que todos los tests fallen (igualando a 0 todos los checkers del test) sin modificar el diseño. A continuación se muestra como se ha igualado el checker de la instrucción ADDI:

```
`CHECK_EQUAL(MEM[SENYALES.INSTRUCCION[11:7]], 0,"Falla en ADDI");
```

Repitiendo el mismo patrón en todos los tests (a excepción del test del reset, en este se ha igualado a un valor distinto de 0) se observa que todas las instrucciones fallan:

```
==== Summary ==================================================
fail tb.tb_top_duv.RESET                   (3.3 seconds)
fail tb.tb_top_duv.ADDI                    (3.1 seconds)
fail tb.tb_top_duv.SLTI                    (2.4 seconds)
fail tb.tb_top_duv.SLTIU                   (2.3 seconds)
fail tb.tb_top_duv.ANDI                    (2.3 seconds)
fail tb.tb_top_duv.ORI                     (2.4 seconds)
fail tb.tb_top_duv.XORI                    (2.3 seconds)
fail tb.tb_top_duv.ADD                     (2.1 seconds)
fail tb.tb_top_duv.SUB                     (2.2 seconds)
fail tb.tb_top_duv.SLT                     (2.2 seconds)
fail tb.tb_top_duv.SLTU                    (2.2 seconds)
fail tb.tb_top_duv.AND                     (2.2 seconds)
fail tb.tb_top_duv.OR                      (2.2 seconds)
fail tb.tb_top_duv.XOR                     (2.1 seconds)
fail tb.tb_top_duv.BEQ                     (2.4 seconds)
fail tb.tb_top_duv.BNE                     (2.3 seconds)
fail tb.tb_top_duv.LW                      (2.4 seconds)
fail tb.tb_top_duv.SW                      (2.3 seconds)
fail tb.tb_top_duv.LUI_opcional            (2.1 seconds)
fail tb.tb_top_duv.AUIPC_opcional          (2.1 seconds)
fail tb.tb_top_duv.JAL_opcional            (2.1 seconds)
fail tb.tb_top_duv.JALR_opcional           (2.4 seconds)
fail tb.tb_top_duv.BLT_opcional            (2.4 seconds)
fail tb.tb_top_duv.BLTU_opcional           (2.3 seconds)
fail tb.tb_top_duv.BGE_opcional            (2.3 seconds)
fail tb.tb_top_duv.BGEU_opcional           (2.4 seconds)
fail tb.tb_top_duv.SLL_opcional            (2.2 seconds)
fail tb.tb_top_duv.SRL_opcional            (2.2 seconds)
fail tb.tb_top_duv.SRA_opcional            (2.1 seconds)
fail tb.tb_top_duv.SLLI_opcional           (2.1 seconds)
fail tb.tb_top_duv.SRLI_opcional           (2.6 seconds)
fail tb.tb_top_duv.SRAI_opcional           (2.8 seconds)
fail tb.tb_top_duv.TODAS_LAS_INSTRUCCIONES (2.1 seconds)
===============================================================
pass 0 of 33
fail 33 of 33
===============================================================
Total time was 76.9 seconds
Elapsed time was 77.0 seconds
===============================================================
Some failed!
```

Figura 4.9: Fallo en todos los tests provocado intencionalmente

Una vez comprobado que los checkers están bien puestos toca comprobar las visualizaciones de onda de todas las instrucciones. Para ello se visualiza la instrucción ADD en formas de onda:

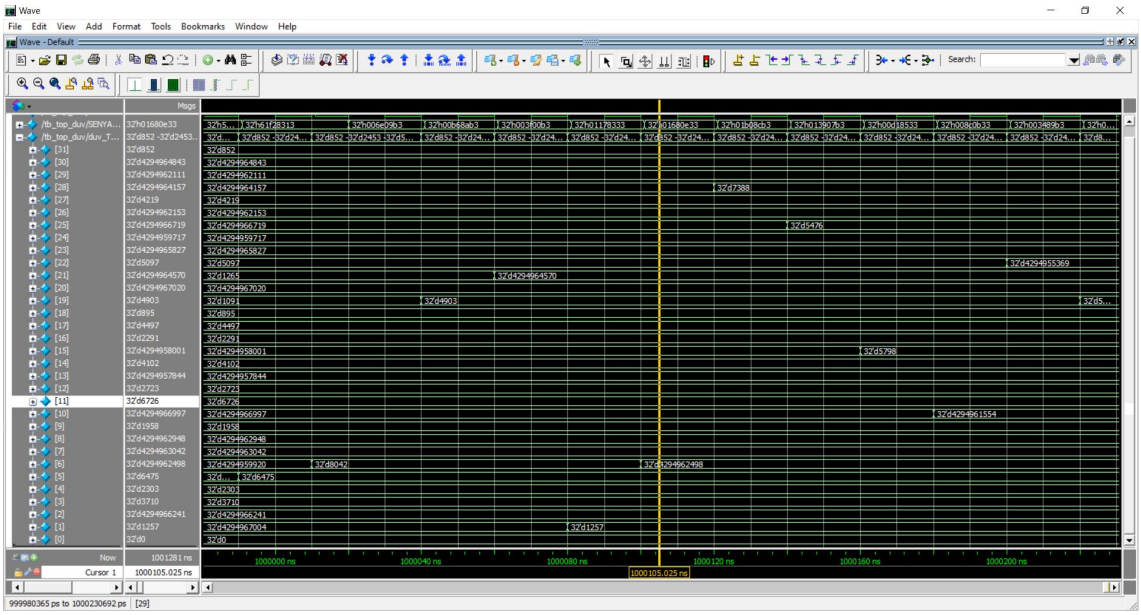

Figura 4.10: Formas de onda instrucción ADD

La instrucción ADD es la que se detalla en la barra amarilla de la propia imagen: 01680e33 (en hexadecimal).

Analizando la instrucción siguiendo las tablas del apartado 2 de este documento la instrucción es la siguiente:

```
add  x28 , x22 , x16
```

- Valor del registro 22: 5097
- Valor del registro 16: 2291
- Valor del registro 28: 4294964157
- Valor del registro 28 tras un ciclo de reloj: 5097 + 2291 = 7338. El valor que se observa en la imágen anterior.

Se puede concluir este apartado comentando que el código realiza la verificación correctamente.

También se ha comprobado con otro microprocesador y la verificación funciona correctamente.

Capítulo 5

Generación de una interfaz gráfica con PyQT

Tras realizar una verificación funcional y exhaustiva era bastante incómodo para los usuarios tener que escribir el comando correspondiente en la terminal para ejecutar una verificación concreta. Así que el proximo paso era realizar una interfaz de usuario mediante PyQT aprovechando la propia interfaz que nos proporciona VUnit.

5.1. ¿Qué es PyQT?

PyQT es la unión de la biblioteca QT a Python que nos permite crear interfaces gráficas ya sea de manera gráfica o programando. Para la realización de este proyecto se decidió realizarlo mediante la programación. A continuación se muestra la interfaz gráfica (diseñada únicamente con PyQT) realizada para este proyecto:

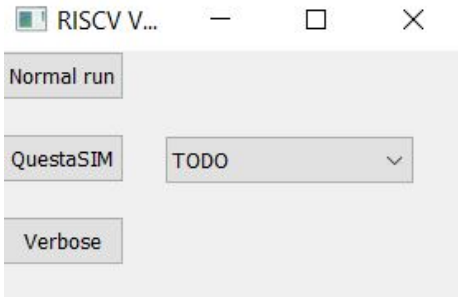

Figura 5.1: Interfaz gráfica del proyecto de PyQT

En está aplicación se decidió realizar uso de la propia interfaz integrada con la terminal proporcionada por VUnit, quedando la interfaz completa como se muestra a continuación:

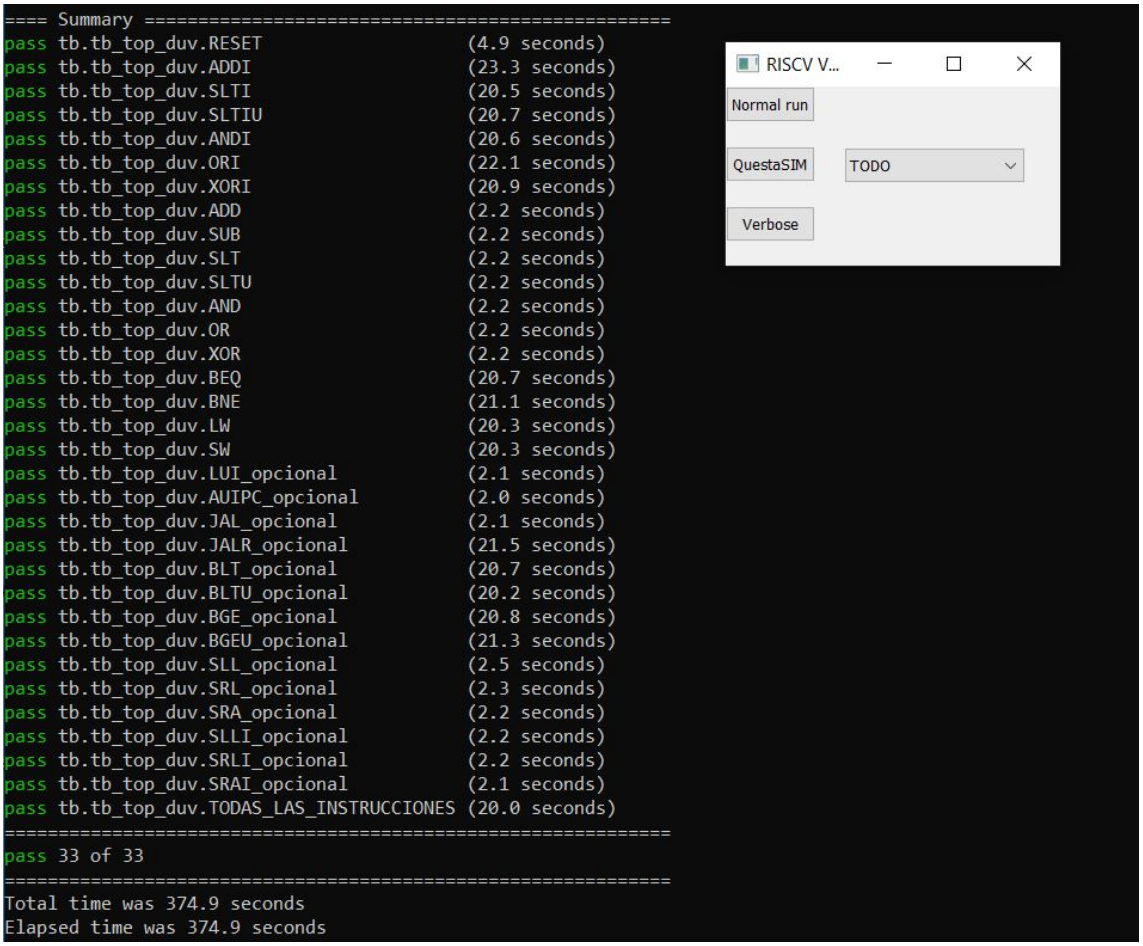

Figura 5.2: Interfaz gráfica del proyecto

5.2. ¿Como se instala PyQT?

Su instalación[5] es muy sencilla, se instala igual que cualquier extensión de Python con el comando ”pip install”. Concretamente se emplea el siguiente comando:

```
pip install PyQt5
```

5.3. Realización de la interfaz gráfica

En un primer instante se optó por obtener ejemplos en la red de una interfaz gráfica con un botón y una lista de selección completamente operativas. Finalmente se acabo encontrando dicho ejemplo en la red.[6]

Se puede apreciar que la ventana de usuario es muy grande, se optó por reducir la misma, así que se decidieron modificar los valores de la misma hasta obtener un tamaño adecuado.

Posteriormente se decidió modificar el tamaño de la lista de selección añadiendo todos los elemen-

tos necesarios, siendo estos todos los tests de verificación por separado, una verificación completa de todos los tests, una verificación con todas las instrucciones obligatorias y una verificación con todas las instrucciones opcionales.

Una vez modificada la lista de selección se procedió a crear dos botones más para introducir las características más demandadas a la hora de configurar un test: abrir el test con el simulador de QuestaSIM y realizar la opción de verbose.

Por último se crearon las acciones de los botones que hacían referencia a cada instrucción con las opciones adecuadas: sin ninguna opción, abrir el simulador de QuestaSIM o simular con verbose.

El código de la interfaz de usuario se muestra a continuación:

```
from PyQt5.QtWidgets import *
from PyQt5 import QtCore, QtGui
from PyQt5.QtGui import *
from PyQt5.QtCore import *
import sys
import os

os.system('"run.py --compile"')
class Window(QMainWindow):

    def __init__(api):
        super().__init__()

        # setting title
        api.setWindowTitle("RISCV VUnit ")

        # setting geometry
        api.setGeometry(300, 300, 280, 150)

        # calling method
        api.UiComponents()

        # showing all the widgets
        api.show()

    # method for widgets
    def UiComponents(api):

        # creating a combo box widget
        api.combo_box = QComboBox(api)

        # setting geometry of combo box
        api.combo_box.setGeometry(100, 52, 150, 28)

        # list of instructions
        lista_comandos = ["TODO","OBLIGATORIAS", "OPCIONALES","TODAS A LA VEZ",
    "RESET", "ADDI", "SLTI", "SLTIU", "ANDI", "ORI", "XORI", "ADD",
        "SUB", "SLT", "SLTU", "AND", "OR", "XOR", "BEQ", "BNE", "LW", "SW", "
    LUI", "AUIPC", "JAL", "JALR", "BLT", "BLTU",
        "BGE", "BGEU", "SLL", "SRL", "SRA", "SLLI", "SRLI", "SRAI"]

        # adding list of items to combo box
        api.combo_box.addItems(lista_comandos)

        # creating push button
```

```
        run_normal = QPushButton("Normal run", api)

        run_normal.setGeometry(0,0,75,30)

        run_gui = QPushButton("QuestaSIM", api)

        run_gui.setGeometry(0,50,75,30)

        run_verbose = QPushButton("Verbose", api)

        run_verbose.setGeometry(0,100,75,30)


        # adding action to the button
        run_normal.pressed.connect(api.normal_run)
        run_gui.pressed.connect(api.questa_run)
        run_verbose.pressed.connect(api.verbose_run)

# create pyqt5 app
App = QApplication(sys.argv)

# create the instance of our Window
window = Window()

# start the app
sys.exit(App.exec())
```

Posteriormente observamos el código reducido de la acción de un botón, el cual, dependiendo del elemento seleccionado realiza una acción u otra:

```
    def questa_run(api):

        if api.combo_box.currentIndex() == 0:
            os.system('"run.py --gui"')

        if api.combo_box.currentIndex() == 1:
            os.system('"run.py tb.tb_top_duv.RESET tb.tb_top_duv.ADDI tb.
    tb_top_duv.SLTI tb.tb_top_duv.SLTIU tb.tb_top_duv.ANDI tb.tb_top_duv.ORI tb
    .tb_top_duv.XORI tb.tb_top_duv.ADD tb.tb_top_duv.SUB tb.tb_top_duv.SLT tb.
    tb_top_duv.SLTU tb.tb_top_duv.AND tb.tb_top_duv.OR tb.tb_top_duv.XOR tb.
    tb_top_duv.BEQ tb.tb_top_duv.BNE tb.tb_top_duv.LW tb.tb_top_duv.SW --gui"')

        if api.combo_box.currentIndex() == 2:
            os.system('"run.py tb.tb_top_duv.RESET tb.tb_top_duv.LUI_opcional
    tb.tb_top_duv.AUIPC_opcional tb.tb_top_duv.JAL_opcional tb.tb_top_duv.
    JALR_opcional tb.tb_top_duv.BLT_opcional tb.tb_top_duv.BLTU_opcional tb.
    tb_top_duv.BGE_opcional tb.tb_top_duv.BGEU_opcional tb.tb_top_duv.
    SLL_opcional tb.tb_top_duv.SRL_opcional tb.tb_top_duv.SRA_opcional tb.
    tb_top_duv.SLLI_opcional tb.tb_top_duv.SRLI_opcional tb.tb_top_duv.
    SRAI_opcional --gui"')

        if api.combo_box.currentIndex() == 3:
            os.system('"run.py tb.tb_top_duv.TODAS_LAS_INSTRUCCIONES --gui"')

        if api.combo_box.currentIndex() == 4:
            os.system('"run.py tb.tb_top_duv.RESET --gui"')

        if api.combo_box.currentIndex() == 5:
```

```
            os.system('"run.py tb.tb_top_duv.ADDI --gui"')

        if api.combo_box.currentIndex() == 6:
            os.system('"run.py tb.tb_top_duv.SLTI --gui"')

        if api.combo_box.currentIndex() == 7:
            os.system('"run.py tb.tb_top_duv.SLTIU --gui"')

        if api.combo_box.currentIndex() == 8:
            os.system('"run.py tb.tb_top_duv.ANDI --gui"')

        if api.combo_box.currentIndex() == 9:
            os.system('"run.py tb.tb_top_duv.ORI --gui"')

        if api.combo_box.currentIndex() == 10:
            os.system('"run.py tb.tb_top_duv.XORI --gui"')

        if api.combo_box.currentIndex() == 11:
            os.system('"run.py tb.tb_top_duv.ADD --gui"')

        if api.combo_box.currentIndex() == 12:
            os.system('"run.py tb.tb_top_duv.SUB --gui"')

        if api.combo_box.currentIndex() == 13:
            os.system('"run.py tb.tb_top_duv.SLT --gui"')

        if api.combo_box.currentIndex() == 14:
            os.system('"run.py tb.tb_top_duv.SLTU --gui"')

        if api.combo_box.currentIndex() == 15:
            os.system('"run.py tb.tb_top_duv.AND --gui"')

        if api.combo_box.currentIndex() == 16:
            os.system('"run.py tb.tb_top_duv.OR --gui"')

        if api.combo_box.currentIndex() == 17:
            os.system('"run.py tb.tb_top_duv.XOR --gui"')

        if api.combo_box.currentIndex() == 18:
            os.system('"run.py tb.tb_top_duv.BEQ --gui"')

        if api.combo_box.currentIndex() == 19:
            os.system('"run.py tb.tb_top_duv.BNE --gui"')

        if api.combo_box.currentIndex() == 20:
            os.system('"run.py tb.tb_top_duv.LW --gui"')

        if api.combo_box.currentIndex() == 21:
            os.system('"run.py tb.tb_top_duv.SW --gui"')

        if api.combo_box.currentIndex() == 22:
            os.system('"run.py tb.tb_top_duv.LUI_opcional --gui"')

        if api.combo_box.currentIndex() == 23:
            os.system('"run.py tb.tb_top_duv.AUIPC_opcional --gui"')

        if api.combo_box.currentIndex() == 24:
            os.system('"run.py tb.tb_top_duv.JAL_opcional --gui"')
```

```
        if api.combo_box.currentIndex() == 25:
            os.system('"run.py tb.tb_top_duv.JALR_opcional --gui"')

        if api.combo_box.currentIndex() == 26:
            os.system('"run.py tb.tb_top_duv.BLT_opcional --gui"')

        if api.combo_box.currentIndex() == 27:
            os.system('"run.py tb.tb_top_duv.BLTU_opcional --gui"')

        if api.combo_box.currentIndex() == 28:
            os.system('"run.py tb.tb_top_duv.BGE_opcional --gui"')

        if api.combo_box.currentIndex() == 29:
            os.system('"run.py tb.tb_top_duv.BGEU_opcional --gui"')

        if api.combo_box.currentIndex() == 30:
            os.system('"run.py tb.tb_top_duv.SLL_opcional --gui"')

        if api.combo_box.currentIndex() == 31:
            os.system('"run.py tb.tb_top_duv.SRL_opcional --gui"')

        if api.combo_box.currentIndex() == 32:
            os.system('"run.py tb.tb_top_duv.SRA_opcional --gui"')

        if api.combo_box.currentIndex() == 33:
            os.system('"run.py tb.tb_top_duv.SLLI_opcional --gui"')

        if api.combo_box.currentIndex() == 34:
            os.system('"run.py tb.tb_top_duv.SRLI_opcional --gui"')

        if api.combo_box.currentIndex() == 35:
            os.system('"run.py tb.tb_top_duv.SRAI_opcional --gui"')
```

5.4. Problemas en la interfaz de usuario

Tras realizar la aplicación hay que comprobar si esta funciona como es debido, para abrirla tenemos que abrir la terminal y ejecutar el siguiente comando desde el directorio donde se encuentre el archivo de PyQT:

```
python ejecutable.py
```

Como se observa, la interfaz de usuario funciona correctamente, se realizan las instrucciones y se puede comprobar en la terminal (interfaz de VUnit) si han pasado la verificación o no (para esta comprobación se ha modificado el diseño para fallar en algunas instrucciones):

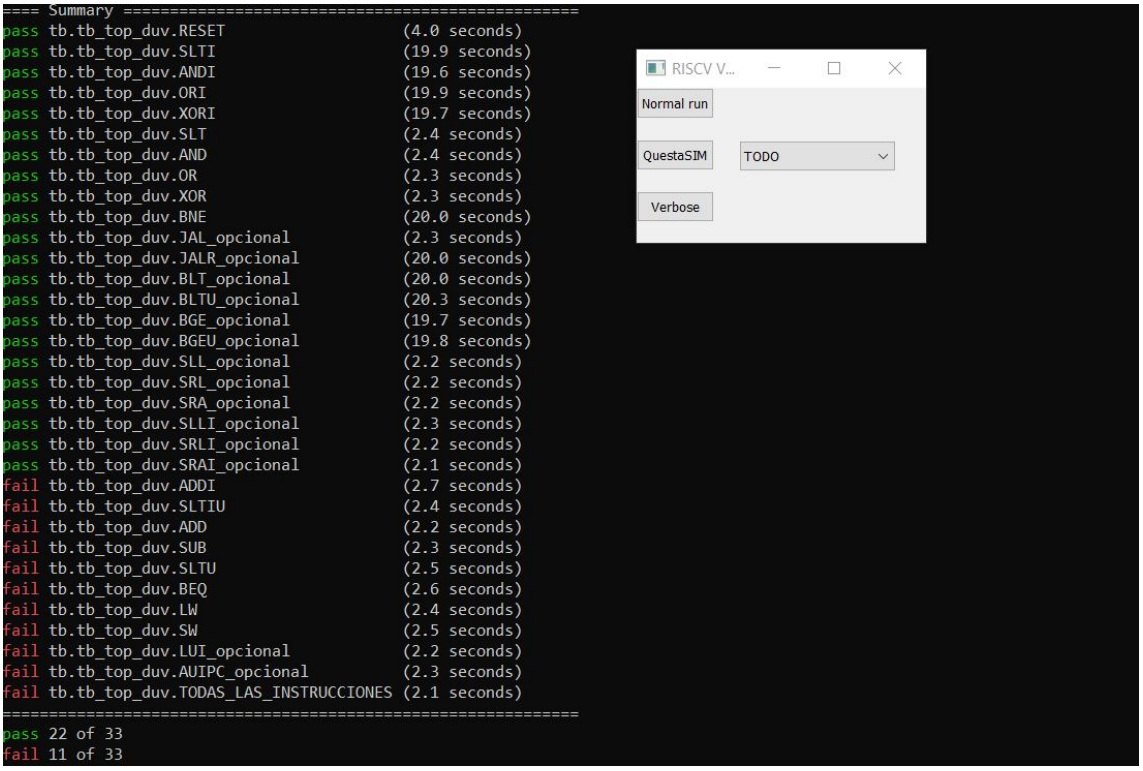

Figura 5.3: Test de la interfaz gráfica modificando partes del diseño

Al final con la interfaz de usuario se ha solucionado parte del problema, los usuarios ya no necesitan reescribir los comandos necesarios para ejecutar el test que quieren realizar, pero siguen teniendo que ejecutar un solo comando y esto dificulta la inmediatez para realizar una verificación.

5.4.1. Solución

La solución es muy sencilla, hay que realizar un archivo ejecutable para poder ejecutar la propia interfaz de usuario. Este archivo tendrá la extensión .bat y contendrá el siguiente código:

```
start cmd /k python ejecutable.py
```

Por último se crea un acceso directo de este archivo y ya podemos tener nuestra propia verificación en el escritorio para poder verificar las instrucciones RV32I de un RiscV

5.5. ¿Que se encuentra alguien que quiere verificar su propio RiscV?

Cualquiera que quiera verificar su propio RiscV recibirá una carpeta comprimida con los siguientes archivos:

- Carpeta _pycache_: No emplear
- Carpeta output_file: Donde se guardaran las instrucciones y los posibles errores causados por instrucciones en la segmentación

- Carpeta src: En esta carpeta se guardaran los archivos de diseño del microprocesador basado en RiscV
- Carpeta vunit_out: No emplear
- Archivo de python, ejecutable: Contiene el código referido a la interfaz de usuario
- Archivo bat, execute: Ejecuta la interfaz de usuario, se debe crear un acceso directo del mismo.
- Archivo de texto, readme: Contiene todas las instrucciones para emplear la verificación.
- Archivo de python, run: Ejecuta el propio test.
- Archivo de System-Verilog: tb_top_duv

Cualquier usuario observe esto, lo primero que realizara será abrir el propio archivo llamado readme el cual contiene la siguiente información:

```
Requisitos:

Python 3.8.5
QuestaSIM

Instalación:

pip install vunit_hdl
pip install -U vunit_hdl
pip install PyQt5

Si no funciona, introducir lo siguiente y repetir la instalación:

pip install wheel

Introducir dos variables de entorno:

VUNIT_SIMULATOR: modelsim
VUNIT_MODELSIM_PATH: dirección de QuestaSIM

Carpeta src:

Seguir las instrucciones del top
Introducir todo tu diseño en esa carpeta, excluyendo el top
NO INCLUIR ROM NI RAM

Ejecutar mediante el archivo .bat llamado execute

-Verbose: da más información sin abrir el QuestaSIM

Salida de las instrucciones generadas en la carpeta: output_file
```

Capítulo 6

Conclusiones y propuesta a futuro

6.1. Conclusión

Primero, cabe recordar el objetivo de este proyecto, verificar un core de un microprocesador mediante VUnit.

El objetivo de la verificación es comprobar si el producto cumple las especificaciones deseadas antes de ser lanzado al mercado para eliminar las pérdidas económicas que puede generar un producto con fallos.

En este proyecto se ha realizado una verificación compleja con RCV implementando el proyecto de la asignatura de ISDIGI y posteriormente se ha implementado en una interfaz de usuario. A continuación se detallan de las tareas a desarrollar en este proyecto cuales se han alcanzado y cuales no:

- Búsqueda de información sobre VUnit y RISC-V. Alcanzado
- Instalación de VUnit. Alcanzado
- Verificación sencilla mediante VUnit. Alcanzado
- Familiarizarse con el entorno de VUnit. Alcanzado
- Implementación del proyecto de ISDIGI. Alcanzado
- Verificación funcional de proyecto. Alcanzado
- Random Constraint Verification (RCV). Alcanzado
- Creación de un top genérico. Alcanzado
- Desarrollo de una interfaz de usuario. Alcanzado
- Utilizar otros procesadores. Alcanzado

Para finalizar el estudio de VUnit: es un método de verificación que no sustituye otros métodos de verificación ya establecidos, sino que es un apoyo para realizar verificaciones más rápidas como las presentadas en este proyecto.

6.2. Propuesta a futuro

A lo largo de este proyecto se han observado dos posibles propuestas a futuro como continuación de este proyecto:

- Verificar un microprocesador segmentado. Tal y como se ha comentado en este documento, sería el siguiente microprocesador a verificar empleando el microprocesador single-cycle como modelo de referencia a verificar.
- Realizar un algoritmo que evalúe si un conjunto de instrucciones no contiene ningún bucle infinito. Cuando se realiza la segmentación, lo ideal es realizar una verificación con un conjunto de instrucciones finito (sin bucles infinitos), para ello habría que realizar una evaluación del conjunto de instrucciones a introducir y observar si es finito.

Bibliografía

[1] *RISC-V*. URL: https://es.wikipedia.org/wiki/RISC-V (visitado 29-08-2022).

[2] *RISC-V Reference Data Card ("Green Card")*. URL: https://inst.eecs.berkeley.edu/~cs61c/fa17/img/riscvcard.pdf (visitado 29-08-2022).

[3] sphinx-quickstart. *riscv-isa-pages documentation master file*. URL: https://msyksphinz-self.github.io/riscv-isadoc/html/rvi.html (visitado 29-08-2022).

[4] Lars Asplund. *VUnit*. URL: https://vunit.github.io/index.html (visitado 29-08-2022).

[5] *stackoverflow, ¿Como instalar y usar PyQt5 en Windows?* URL: https://es.stackoverflow.com/questions/67995/como-instalar-y-usar-pyqt5-en-windows (visitado 29-08-2022).

[6] *GeeksForGeeks, PyQt5 – Show pop up items of ComboBox when push button is pressed*. URL: https://www.geeksforgeeks.org/pyqt5-show-pop-up-items-of-combobox-when-push-button-is-pressed/ (visitado 29-08-2022).